Homework 9: Deterministic Execution

Intro

- [ ] Why would a bug happen only on awsrun but not on your development machine? Why would code that works great for you, occasionally crash for your partner, and always for your TA?

▼ **Ans**

"——These are a derivative of environmental, time, and randomness sources beyond our control."

```
int main() {
    int i;
    printf("value of i=%d\n", i);
    printf("address &i=%p\n", &i);
    printf("hash of i=%ld\n", ((uintptr_t)&i) & 127);
    return 0;
}
```

- [ ] Run it a few times using **make undef-compare** and compare the results. Are they the same? Why or why not?

▼ **Ans**

```
make CFLAGS=-O3 clean undef; ./undef
value of i=32765
address &i=0x7ffd706bb8b4
hash of i=52

make CFLAGS=-O1 clean undef; ./undef
value of i=32766
address &i=0x7ffef79af184
hash of i=4
```

- [ ] Fix the code to define the variable and rerun **make undef-compare** to make sure that it worked.

▼ **Ans**

```
make CFLAGS=-O3 clean undef; ./undef
value of i=100
address &i=0x7ffd20f06a34
hash of i=52

make CFLAGS=-O1 clean undef; ./undef
value of i=100
address &i=0x7ffcbd6e4484
hash of i=4
```

- [ ] Run **make undef-noaslr** to run the program a few times without ASLR. Is this program deterministic now?

▼ **Ans**

```
setarch x86_64 -R ./undef
value of i=100
address &i=0x7fffffffd734
hash of i=52

setarch x86_64 -R ./undef
value of i=100
address &i=0x7fffffffd734
hash of i=52
```

- [ ] Run **make undef-env** to run the command with differing environments. What is printed now?

▼ **Ans**

```
setarch x86_64 -R env USER=me ./undef
value of i=100
address &i=0x7fffffffd734
hash of i=52

setarch x86_64 -R env USER=professoramarasinghe ./undef
value of i=100
address &i=0x7fffffffd724
hash of i=36
```

Checkoff Item 1

▼ **Hashtable**

slot	ptr	size
0		
1		
2		
3		
4		
5		
6		

☐ What do you need to make this program deterministic? Modify the Makefile target for **hashtable1000-good** so that all runs succeed.

```
# XXX modify this target so that 1000 runs all succeed
hashtable1000-good: ./hashtable
    bash -c "for((i=0;i<1000;i++)) ./hashtable; done"
```

▼ **Ans**

```
bash -c "for((i=0;i<1000;i++)) do setarch x86_64 -R ./hashtable 1 0; done" // always
 succeed
```

Checkoff Item 2

☐ Modify **hashtable1-bad** target so the program always fails. Show your modified Makefile target. You may find it useful to examine **hashtable.c** to see what system arguments it takes.

```
# XXX modify this target so the program always fails
hashtable1-bad: ./hashtable
    ./hashtable
```

▼ **Ans**

```
bash -c "for((i=0;i<1;i++)) do setarch x86_64 -R ./hashtable 1 5; done" // always fai
l
```

Checkoff Item 3

- [ ] What was the bug? What is your fix? Rerun make **hashtable1000** to ensure that your fix always works.

```
void hashtable_insert(void* p, int size) {
  assert(ht.entries < TABLESIZE);
  ht.entries++;

  int s = hash_func(p);
  /* open addressing with linear probing */
  do {
    if (!ht.hashtable[s].ptr) {
      ht.hashtable[s].ptr = p;
      ht.hashtable[s].size = size;
      break;
    }
    /* conflict, look for next item */
    s++;
  } while (1);
}
```

▼ **Ans**

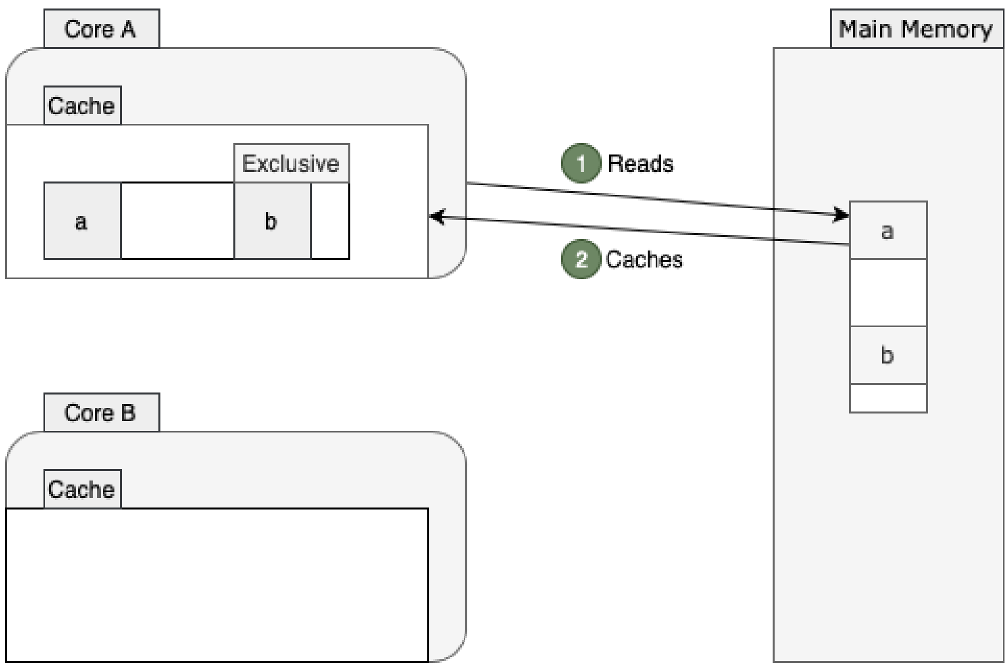
```
void hashtable_insert(void* p, int size) {
  hashtable_lock();
  assert(ht.entries < TABLESIZE);
  /* ensure atomic, can also use std::atomic<int> */
  ht.entries++;
  int s = hash_func(p);
  /* open addressing with linear probing */
  do {
    if (!ht.hashtable[s].ptr) {
      ht.hashtable[s].ptr = p;
      ht.hashtable[s].size = size;
      break;
    }
    /* conflict, look for next item */
    s++;
```

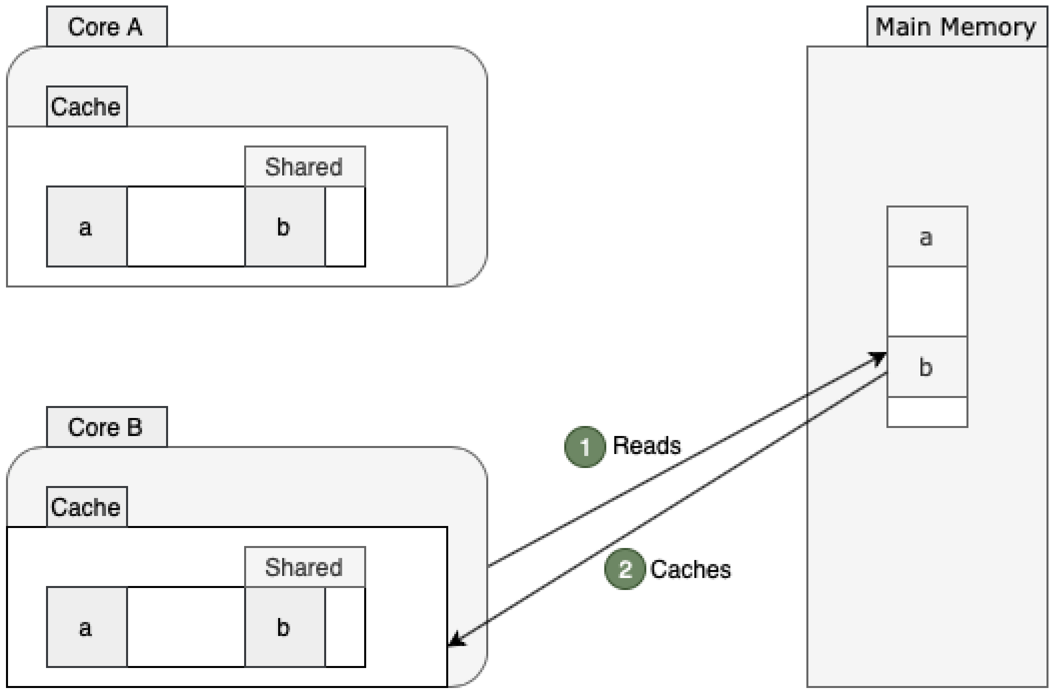
```
    /* ensure the slot can be reset */
    s %= TABLESIZE;
  } while (1);
  hashtable_unlock();
}
```

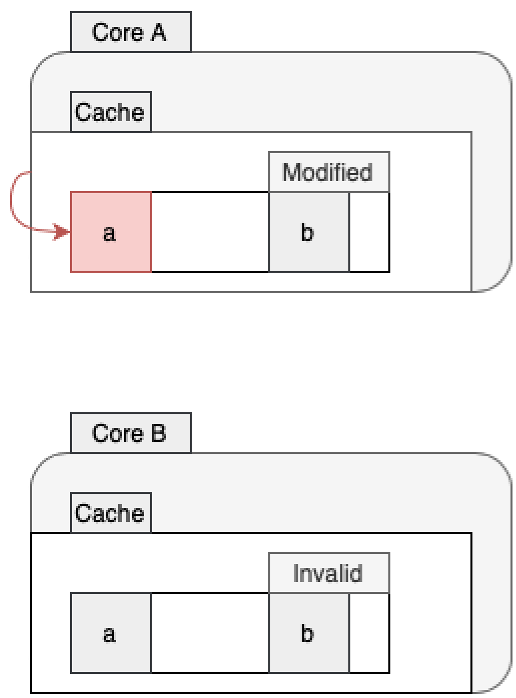
Write-up 1

- [ ] Is the hashlock implementation in **hashlock.c** vulnerable to false sharing? Why or why not?

▼ **false sharing**







▼ **Ans**

```
// ori entry
typedef struct entry {
```

```
  void* ptr; // 8 bytes
  size_t size; // 8 bytes
} entry_t;

// 64 bytes alignas entry
struct alignas(64) entry_t {
  void* ptr;
  size_t size;
};

// ori hashlocks
pthread_mutex_t hashlock[HLOCKS];
#define HASHLOCK_SLOT(l) &hashlock[(l) & (HLOCKS - 1)]

// 64 bytes alignas hashlocks
struct alignas(64) my_lock {
    pthread_mutex_t my_hash_lock;
}
my_lock[HLOCKS];
#define MY_HASHLOCK_SLOT(l) &(my_lock[(l) & (HLOCKS - 1)].my_hash_lock)

// Ans
// In theory yes, but not in practice.
```

Write-up 2

- [ ] What problem is the fairness solution in **hashtable_insert_fair** introducing? Explain or demonstrate the behavior you see.

▼ **Ans**

hashtable_insert_fair locks and unlocks much more often than **hashtable_insert_locked** when keys of all test cases conflict, which is a huge cost and the actual measured time difference is tens of times larger.

10 threads * 20	all same keys (cost time/lock times/conflict times)	all diff keys (cost time/lock times/conflict times)
hashtable_insert_locked	0.5~3ms / 200 / 19900	0.5~3ms / 200 / 0
hashtable_insert_fair	100ms / 20300 / 19900	0.5~3ms / 400/ 0

> Ps: 200 = 10 threads * 20   20300 = 200 + (1 + 200) * 200 / 2
> 19900 = (1 + 199) * 199/ 2

Write-up 3

- [ ] Use InterlockedCompareExchange64 to implement your changes in **hashtable_insert_lockless**. Don't forget to modify **hashtable_fill** to use your new code. Run make **hashtable-mt1000** to ensure that it works. What changes are necessary in **hashtable_lookup** so you can use just a 8-byte compare-and-swap in **hashtable_insert_lockless**?

```
static inline uint64_t InterlockedCompareExchange64(volatile uint64_t* ptr, uint64_t _new,
                                                    uint64_t old) {
  uint64_t prev;
  asm volatile("lock;"
               "cmpxchgq %1, %2;"
               : "=a"(prev)
               : "q"(_new), "m"(*ptr), "a"(old)
               : "memory");
  return prev;
}
```

▼ **CAS(Compare & Swap)**

```
uint64_t CAS(uint64_t* ptr, uint64_t _new, uint64_t old) {
    if (*ptr == old) {
        *ptr = _new;
    } else {
        old = *ptr;
    }
    return old;
}
```

▼ **Ans**

```
void hashtable_insert_lockless(void* p, int size, int tid) {
    hashtable_lock();
    assert(ht.entries < TABLESIZE);
    ht.entries++;
    hashtable_unlock();

    int s = hash_func(p);
    printf("hashtable_insert_lockless: tid:%d slot:%d\n", tid, s);
    /* open addressing with linear probing */
    do {
    if (!InterlockedCompareExchange64((uint64_t*)&ht.hashtable[s].ptr, (uint64_t)p,
 0)) {
        ht.hashtable[s].size = size;
        break;
```

```
    }
    s++;
    s %= TABLESIZE;
    } while (1);
}
```

Write-up 4

- [ ] (No implementation required.) How would you implement a deterministic hashtable structure, such that the order of insertions does not change the final hashtable state?

▼ **Ans**

```
#include <iostream>
#include <stdio.h>
#include <typeinfo>
#include <thread>
#include <vector>
#include <mutex>
#include <atomic>
#include <assert.h>

using namespace std;

#define TABLESIZE 256
#define THREADS 10

struct ListNode {
    int ptr;
    int size;
    ListNode* next;
    ListNode() : ptr(0), size(0), next(nullptr) {}
    ListNode(int p, int s) : ptr(p), size(s), next(nullptr) {}
    ListNode(int p, int s, ListNode* n) : ptr(p), size(s), next(n) {}
};

typedef struct hashtable_t {
    ListNode* head_node[TABLESIZE];
    int entries;
    pthread_mutex_t lock;
} hashtable_t;

int hash_func(int ptr) {
    return ptr / 10;
}

void is_greater(int ptr, ListNode* cur_node, bool& res) {
    assert(cur_node);
    res = ptr > cur_node->ptr;
}
```

```
hashtable_t ht = {{}, 0, PTHREAD_MUTEX_INITIALIZER};

void hashtable_lock() {
  pthread_mutex_lock(&ht.lock);
}

void hashtable_unlock() {
  pthread_mutex_unlock(&ht.lock);
}

void hash_insert(int ptr, int size) {
    hashtable_lock();
    assert(ht.entries < TABLESIZE);
    ht.entries++;

    int slot = hash_func(ptr);
    if (ht.head_node[slot] == nullptr) {
        ht.head_node[slot] = new ListNode(ptr, size);
    } else {
        bool res = true;
        ListNode* cur_node = ht.head_node[slot];
        ListNode* pre_node = nullptr;
        while (cur_node) {
            is_greater(ptr, cur_node, res);
            if (!res) break;
            pre_node = cur_node;
            cur_node = cur_node->next;
        }
        ListNode* ptr_node = new ListNode(ptr, size);
        if (res) {
            pre_node->next = ptr_node;
        } else {
            if (pre_node) {
                ptr_node->next = cur_node;
                pre_node->next = ptr_node;
            } else {
                ptr_node->next = cur_node;
                ht.head_node[slot] = ptr_node;
            }
        }
    }
    hashtable_unlock();
}

int main(int argc, char* argv[]) {
    // hash_insert(13, 100);
    // hash_insert(15, 100);
    // hash_insert(11, 100);

    int group = 0;
    if (argc > 1) {
        group = atoi(argv[1]);
    }
```

```
    vector<shared_ptr<thread>> vec_ptr;
    for (int i = 0; i < THREADS; i++) {
        vec_ptr.push_back(make_shared<thread>(hash_insert, group * 10 + i, 100));
    }

    for (int i = 0; i < THREADS; i++) {
        vec_ptr.at(i)->join();
    }

    ListNode* cur = ht.head_node[group];
    while (cur) {
        cout<<"key:"<<cur->ptr<<" val:"<<cur->size<<endl;
        cur = cur->next;
    }

    return 0;
}

root@CD-DZ0104843:/home/hanbabang/2_workspace/MIT6_172F18_hw9/MIT6_172F18_hw9# cd "/h
ome/hanbabang/2_workspace/MIT6_172F18_hw9/MIT6_172F18_hw9/" && g++ hashtable_insert_o
blivious_.cpp -pthread -o hashtable_insert_oblivious_ && "/home/hanbabang/2_workspac
e/MIT6_172F18_hw9/MIT6_172F18_hw9/"hashtable_insert_oblivious_ 5
Insert key:50 val:100
Insert key:53 val:100
Insert key:55 val:100
Insert key:52 val:100
Insert key:57 val:100
Insert key:56 val:100
Insert key:54 val:100
Insert key:51 val:100
Insert key:59 val:100
Insert key:58 val:100

Dump   key:50 val:100
Dump   key:51 val:100
Dump   key:52 val:100
Dump   key:53 val:100
Dump   key:54 val:100
Dump   key:55 val:100
Dump   key:56 val:100
Dump   key:57 val:100
Dump   key:58 val:100
Dump   key:59 val:100
```